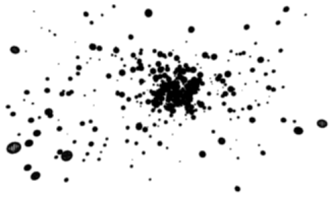
Sorbonne Université – SESI M2

MU5IN160 – Parallel Programming



Project – n-body: Move U'r Body

1 Introduction

In this project, you will work on a classic compute intensive application that simulates the movement of planets as a function of time (= a n-body problem). A working application, called "MUrB", is given to you. You will have to speedup MUrB thanks to the different levels of parallelism and optimization methods you learned in this class.

First you need to clone the repository of the MUrB project:

```
git clone --recursive https://gitlab.lip6.fr/parallel-programming/murb-se.git
```

The MUrB project uses CMake in order to generate a Makefile: follow the README instructions to compile the code.

2 Scientific Background – The Governing Equations

The n-body problem is a classic one from the **Newtonian mechanics**: it consists in **resolving the gravity equations**. However, this problem can be generalized (from the algorithmic point of view) and we can often find it in numerical simulation: this is why it is a good real case study.
At the problem beginning (the time t), for each body i, its position $q_i(t)$, its mass m_i and its velocity $\vec{v_i}(t)$ are known. The applied force between two bodies i and j, at the t time, is defined by:

$$\vec{f_{ij}}(t) = G.\frac{m_i.m_j}{||\vec{r_{ij}}||^2}.\frac{\vec{r_{ij}}}{||\vec{r_{ij}}||}, \tag{1}$$

with G the gravitational constant ($G = 6,67384 \times 10^{-11} m^3.kg^{-1}.s^{-2}$) and $\vec{r_{ij}} = q_j(t) - q_i(t)$ the vector from body i to body j. The resulting force (alias the total force) for a given i body, at the t time, is defined by:

$$\vec{F_i}(t) = \sum_{j \neq i}^{n} \vec{f_{ij}}(t) = G.m_i.\sum_{j \neq i}^{n} \frac{m_j.\vec{r_{ij}}}{||\vec{r_{ij}}||^3}, \tag{2}$$

with n the number of bodies in space. For the time integration, the acceleration is required. For a given body i, at the t time, the acceleration characteristic is defined by:

$$\vec{a_i}(t) = \frac{\vec{F_i}(t)}{m_i} = G.\sum_{j \neq i}^{n} \frac{m_j.\vec{r_{ij}}}{||\vec{r_{ij}}||^3}. \tag{3}$$

2.1 An Approximation of the Total Force

The total force $\vec{F_i}$ is given by Eq. 2:

$$\vec{F_i}(t) = G.m_i.\sum_{j \neq i}^{n} \frac{m_j.\vec{r_{ij}}}{||\vec{r_{ij}}||^3}.$$

As bodies approach each other, **the force between them grows without bound**, which is an undesirable situation for numerical integration. In astrophysical simulations, collisions between bodies are generally precluded; this is reasonable if the bodies represent galaxies that may pass right through each other. Therefore, **a softening factor** $\epsilon^2 > 0$ is added, and the denominator is rewritten as follows:

$$\vec{F_i}(t) \approx G.m_i.\sum_{j=1}^{n} \frac{m_j.\vec{r_{ij}}}{(||\vec{r_{ij}}||^2 + \epsilon^2)^{\frac{3}{2}}}. \tag{4}$$

Note the condition $j \neq i$ is no longer needed in the sum, because $\vec{f_{ii}} = 0$ when $\epsilon^2 > 0$. The softening factor models the interaction between two Plummer point masses: masses that behave as if they were spherical galaxies. In effect, **the softening factor limits the magnitude of the force between the bodies**, which is desirable for numerical integration of the system state.
As before, we need to compute the acceleration in order to perform the integration over the time:

$$\vec{a_i}(t) = \frac{\vec{F_i}(t)}{m_i} \approx G.\sum_{j=1}^{n} \frac{m_j.\vec{r_{ij}}}{(||\vec{r_{ij}}||^2 + \epsilon^2)^{\frac{3}{2}}}. \tag{5}$$

2.2 Time Integration

The integrator used to update the positions and velocities is a leapfrog-Verlet integrator (Verlet 1967) because it is applicable to this problem and is computationally efficient (it has a high ratio of accuracy to computational cost). Note that is this project **only constant Δt time step is considered.**
Body i velocity characteristic at the $t + \Delta t$ time depends on the velocity and the acceleration at the t time:

$$\vec{v_i}(t + \Delta t) = \vec{v_i}(t) + \vec{a_i}(t).\Delta t. \tag{6}$$

At the end, body i position q_i at the $t+\Delta t$ time depends on the position, the velocity and the acceleration at the t time:

$$q_i(t + \Delta t) = q_i(t) + \vec{v_i}(t).\Delta t + \frac{\vec{a_i}(t).\Delta t^2}{2}. \tag{7}$$

Thanks to Eq. 5, 6 and 7, it is now possible to compute the new position and the new velocity for all bodies at the $t + \Delta t$ time.

3 Getting Started

3.1 Architecture of the Project

MUrB is C++ project. The sources and the headers are located in the `src` folder. The latest contains 3 sub-folders:

- `common`: 1) the core interface for the n-body simulation, 2) implementations dedicated to the visualization and 3) some tools for performance measurements and for the management of the command line interface.
- `murb`: 1) the main function of MUrB and 2) different simulation implementations (at the beginning, a working implementation is given as the golden model).
- `test`: the test files to compare simulation implementation results to the golden model results.

Note that you should **never modify the files in the `common` folder**. When you will submit your work, you will only upload the `murb` and `test` folders. However, the `common/core` folder contains two very important files for your work in this project (**you should have a look on them**):

- `SimulationNBodyInterface`: this class defines the interface of the new implementations you will have to create. These implementations will inherit from the `SimulationNBodyInterface` and you will have to implement the pure & virtual `computeOneIteration()` method.
- `Bodies`: a class used by the `SimulationNBodyInterface`. It contains the data of the bodies to simulate (masses, radiuses, positions, velocities and more).

In the proposed architecture, the implementations (that inherit from the `SimulationNBodyInterface` class) are in charge of the allocation, the initialization and the computation of the bodies acceleration (see Eq. 5) while the `Bodies` class is updating the velocities (see Eq. 6) and positions (see Eq. 7) thank to the time integration.

Additionally, the `Bodies.hpp` header defines some useful structures to exchange data between the computation of the accelerations and velocity & position updates. Each time it is possible to choose between two memory layouts: Array of Structures (AoS) and Structure of Arrays (SoA).

3.2 Add New Implementation

First you need to create a new class that inherit from `SimulationNBodyInterface`. This class will be located in the `src/murb/implem` folder. You will put the class definition into a C++ header file (example: `SimulationNBody[ImplName].hpp`) while the implementation will be in a source file (example: `SimulationNBody[ImplName].cpp`). If you have some doubts on how to do this you can have a look on the implementation of the golden model (`SimulationNBodyNaive.hpp` and `SimulationNBodyNaive.cpp` files). You can start by copy-pasting the golden model files and renaming them.

In order to compile the new implementation, you need to re-generate the Makefile to include the new files. For this, you just need to call CMake without any argument. CMake will automatically remember the initial options from its previous invocation and it will produce a Makefile with the new source files:

```
cmake .. && make -j4
```

Next step is to instantiate the new implementation into the `murb/main.cpp` file. First, after line 21, you need to include the corresponding header:

```
#include "implem/SimulationNBody[ImplName].hpp"
```

Then, you need to add a command line tag to select the new implementation (line 76):

```
1  docArgs["-im"] = "code implementation tag:\n"
2                   "\t\t\t - \"cpu+naive\"\n"
3                   "\t\t\t - \"[TagName]\"\n" // <= this is the new tag
4                   "\t\t\t ----";
```

Finally, you need to instantiate the right simulation object when the `[TagName]` tag is specified from the command line (see `createImplem` function line 181):

```
1  SimulationNBodyInterface *createImplem()
2  {
3    SimulationNBodyInterface *simu = nullptr;
4    if (ImplTag == "cpu+naive") {
5      simu = new SimulationNBodyNaive(NBodies, BodiesScheme, Softening);
6    }
7    else if (ImplTag == "[TagName]") {
8      simu = new SimulationNBody[ImplName](NBodies, BodiesScheme, Softening);
9    }
10   else {
11     std::cout << "Implementation '" << ImplTag << "' does not exist... Exiting." << std::endl;
12     exit(-1);
13   }
14   return simu;
15 }
```

Now you can execute MUrB with the new implementation with the following command:

```
./bin/murb -n 1000 -i 1000 -v --im [TagName]
```

Note that in the previous examples, the code relative to the new implementation is highlighted and `[ImplName]`/`[TagName]` are place-orders in the code: replace them by the real names.

3.3 Code Validation

Validation of new implementations is a very important step! Non-validated code won't be taken into account in the project grade. Some tests are given in the `test/implem/dumb.cpp` file (based on the Catch2 test framework). These tests will always pass. Indeed, `simuRef` and `simuTest` are both objects from the same `SimulationNBodyNaive` class. However, this file is intended to be use as a template for new implementations.

Each time you will create a new implementation, you will create a new test in the `test/implem` folder. You can copy-paste the `dumb.cpp` test and rename it. Then you will instantiate the new implementation in the `simuTest` object. This will automatically compare the golden model with the new implementation.

Before to run the tests, if you added new files in the `test/implem` folder, you need to re-generate the Makefile and to re-compile the tests:

```
cmake ..
make -j4
```

Then you can run the tests as follow:

```
./bin/murb-test
```

If you see the following message:

```
===============================================================================
All tests passed (XXXXX assertions in Y test case)
```

you are good to go ;-). Otherwise, it means that there is a problem in your implementation and it is time to debug.

4 Assignments

The main purpose of the project is to write new implementations of the n-body (computation of the bodies acceleration) in order to increase the throughput (the number of FPS). As it is a project, you are free to use any methods you want: be creative! All the new implemented versions will count for the final grade and not only the fastest one. The next section gives you the main axes to guide you a little bit.

4.1 Main Axes

Sequential. The given implementation (`SimulationNBodyNaive`) can be improved: too many operations are performed and the overall complexity can be reduced. In the golden model, the computational complexity is $\mathcal{O}(n^2)$, can we do better than this?
For instance, you can work on a new implementation named `SimulationNBodyOptim` and you can associate this class to the `cpu+optim` tag.

SIMD. Of course, this application can benefit from SIMD instructions. Think about reducing the number of loads and about the cost of the instructions will help you to write a fast implementation. MIPP implementation is preferred over NEON intrinsic functions. NEON version is accepted but for comparable number of FPS, MIPP version is better because the code is portable over most of the recent architectures.
For instance, you can work on a new implementation named `SimulationNBodySIMD` and you can associate this class to the `cpu+simd` tag.

Multi-threading. For the multi-threaded version, you will certainly consider one or both of the previous versions (sequential & optimized and/or SIMD). In any case, please copy-paste the previous codes to create new OpenMP versions. For performance reproducibility, please put some comments just before the OpenMP pragma with the full command line you used. Here is an example of what is expected:

```
OMP_NUM_THREADS=4 OMP_SCHEDULE="static,1" ./bin/murb -n 5000 -i 1000 -v --im cpu+omp
```

For instance, you can work on a new implementation named `SimulationNBodyOpenMP` and you can associate this class to the `cpu+omp` tag.

GPU. For the GPU version it is advised to use OpenCL as we learned it in class but you can also use CUDA or OpenMP if you know what you are doing ;-). Be aware that a CUDA version will only execute on Nvidia GPUs. You can also implement both OpenCL and CUDA versions to compare if it changes something on the achieved FPS.
For instance, you can work on a new implementation named `SimulationNBodyGPUOpenCL` and you can associate this class to the `gpu+ocl` tag.

Heterogeneous. You implemented an efficient n-body version on CPU and on GPU? Why not to combine both of these implementations to achieve even higher FPS?
For instance, you can work on a new implementation named `SimulationNBodyHetero` and you can associate this class to the `hetero` tag.

4.2 Awards

Here is a list of awards that give a bonus on the final grade:

1. "*I'm the fastest in the World!*" +3 points
   For fastest implementation in the class. It will be evaluated with the following command:

   ```
   ./bin/murb -n 30000 -i 10000 -v --nv --im [ImplTag]
   ```

2. "*100%, I'm a good student!*" +2 points
   For the group that will have covered the biggest number of implementations, including the suggested ones. All of these implementations have to be significantly different and must present an interest for the n-body code optimization.

3. "*I'm only the 2^{nd} fastest :'(*" +1.5 point
   Same as the first award but for the second fastest...

4. "*I'm an artist!*" +1 point
   For the group that will have the most original implementation. Something based on a new idea not suggested in this document.

5. "*Open-source forever!*" max +2 points
   Any useful contribution to the MIPP library will give you an extra +0.5 point on the project grade (up to +2 points). This is not a classic award because everyone can participate. Each contribution has to be submitted with a *Pull Request* (PR) on GitHub. Contribution that has already be proposed by an other student before will not give extra point. In other word, check the last version of MIPP before submitting a new contribution.

The achievements cannot be combined, it means that 4 different groups will received one award (except for the award n°5). Note that you can receive an award without meeting the requirements of the main project. It is just for fun!
You may also notice that this is a little bit competitive and this is absolutely not mandatory to have a good grade.

5 Due Report

The report is a document in which you will explain the optimizations you performed and you will show the performance improvement with plots and tables. You can write it in French or in English. But be aware that **the spelling and the care you take with the document will be an important part of the final grade**. It is strongly recommended to adopt the following organization (**in bold**: the most important sections):

1. Introduction / Presentation
2. **Description of the Optimizations**
3. Computer Architecture
4. **Experimentation Results**
5. Conclusion

Introduction is a section where you will present the project. You will explain its main purpose. It is not a "copy-paste" of this document. What is interesting is your vision/understanding of the project.

The description of the optimizations is an important part of the report, we expect that you explain the choices you made to improve the efficiency of the code. Thus, **it is strongly recommended to add some illustrative figures to facilitate the reader's understanding.** You can also add some small source code parts to show a specific technique. But be aware, you should not have too much source code on your report.

Before the experimentation section, you will describe your testbed: the Jetson TX2 architecture (CPU, GPU & memory).

In the experimentation section, you will mainly consider the performance in term of throughput (= FPS). For instance, you can observe:

- FPS depending on the number of cores (= speedup curves),
- FPS depending on the number of bodies,
- FPS depending on the initial version versus the SIMD version,
- FPS depending on the CPU and the GPU versions,
- And so on.

Please note that MUrB comes with a command line argument to disable the visualization (`--nv`): you will use it for the performance measurements. An other important metric is the number of floating point operations per second (see the Gflops with the `--gf` argument). For instance, it could be interesting to observe and to comment the evolution of the Gflops depending on the number of bodies...

In the conclusion, you will recall the most important results of your project, as well as your overall point of view. Finally you will give some directions for future improvements.

List of things to avoid:

- Screenshots: please type the text in the document, images are generally heavy and ugly,
- Unstructured document: use Microsoft Word, Libre Office, LaTeX, ...
- Start to write the report at the last time: you should start as soon as possible and complete it while you are working on the project.

6 Work Submission

You will have to submit both the code and the report on Moodle. For the report, you will submit a "PDF" file named `report.pdf`. For the code, you will send a "zip" file named `code.zip`. The latest will contain only two folders of the project: `src/murb` and `src/test`. Note that you should also include the `CMakeLists.txt` file if you modified it. Please do NOT include binaries or the `src/common` folder.

7 Appendix

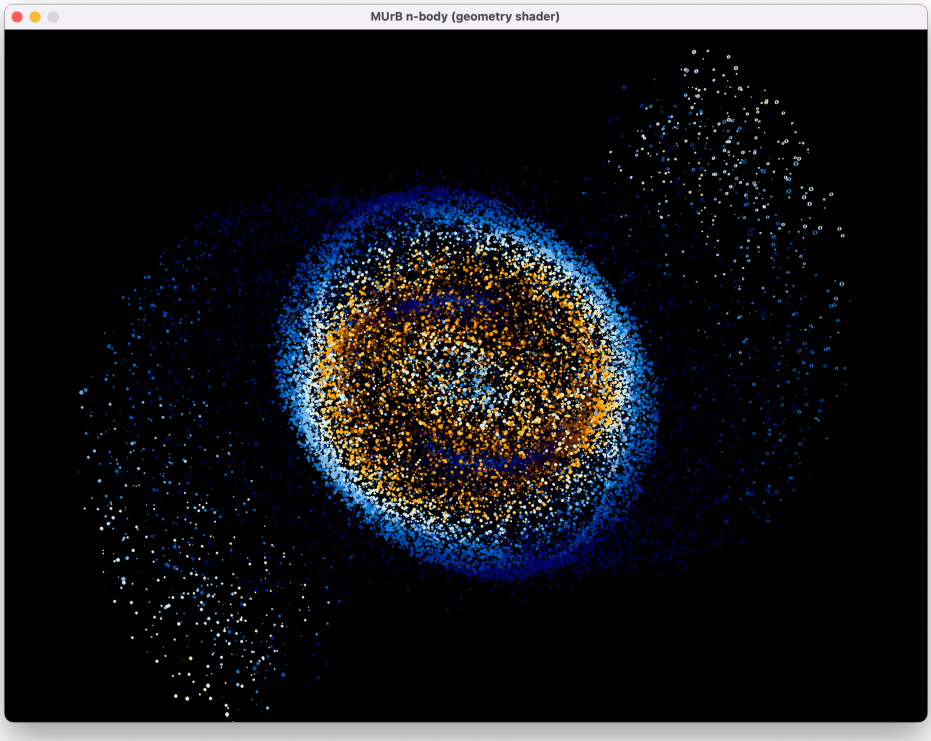


Figure 1: Simulation of a galaxy with 50000 bodies.

Figure 2: Bodies color palette depending on the relative velocity.

Fig. 1 shows an example of galaxy visualization and Fig. 2 gives the corresponding color palette. For each iteration, MUrB finds the slowest body and the fastest body. Then, for each body, its color depends on its velocity relatively to the slowest and fastest body. Note that the colors are just here for aesthetic purpose, their scientific interest is questionable.